# 基于自动机的 Java 信息流分析 *

吴泽智 [1,2]，陈性元 [1,2]，杜学绘 [1]，杨 智 [1]

(1. 解放军信息工程大学 密码工程学院，郑州 450001; 2. 密码科学技术国家重点实验室，北京 100094)

**摘 要：**面向 Java 的信息流分析工作需要修改编译器或实时执行环境，对已有系统兼容性差，且缺乏形式化分析与安全性证明。首先，提出了基于有限状态自动机的 Java 信息流分析方法，将整个程序变量污点取值空间抽象为自动机状态空间，并将 Java 字节码指令看做自动机状态转换动作；然后，给出了自动机转换的信息流安全规则，并证明了在该规则下程序执行的无干扰安全性；最后，采用静态污点跟踪指令插入和动态污点跟踪与控制的方法实现了原型系统 IF-JVM，既不需要获得 Java 应用程序源码，也不需要修改 Java 编译器和实时执行环境，更独立于客户操作系统。实验结果表明，原型系统能正确实现对 Java 的细粒度地信息流跟踪与控制，性能开销为 53.1%。

**关键词：**有限状态自动机；动态污点跟踪；信息流分析；无干扰；Java

**中图分类号：**TP309.1  doi: 10.3969/j.issn.1001-3695.2017.08.0705

## Automata-based information flow analysis for Java

Wu Zezhi[1,2], Chen Xingyuan[1,2], Du Xuehui[1], Yang Zhi[1]

(1. *College of Cryptogram Engineering*, *PLA Information Engineering University*, *Zhengzhou* 450001, *China*; 2. *State Key Laboratory of Cryptology*, *Beijing* 100094, *China*)

**Abstract:** Existing Java-oriented information flow analysis works do not compatible with current systems due to the modifying of the compiler or run-time execution environment. At the same time, they also lack of formal analysis and security proof. First, this paper proposed a formal Java-oriented information flow analysis method based on finite state automata. It abstracted the taint value space of entire program variables into the state space of automata and transferred the Java bytecode instructions into the state transition actions of automata. Then, it given the information flow security rules of state machine conversion and proved the noninterference security property under these rules. Finally, it implemented the prototype system named IF-JVM by using the static taint track instruction inserting and dynamic taint tracking technologies. IF-JVM is independent of the customer operating system. Neither needs to get the source code of Java application, nor needs to modify the Java compiler or run-time execution environment. The experimental results show that the IF-JVM is an accurate system that tracking and controlling information flow for the Java with the 53.1% overhead on performance.

**Key Words:** finite state automata; dynamic taint tracking; information flow analysis; noninterference; Java



## 0 引言

动态信息流跟踪[1]又可称为动态污点跟踪或动态数据流跟踪，指在程序执行过程中跟踪所感兴趣的数据流集合。信息流控制机制实现的核心思想是将标签(附着在数据上，标签随着数据在整个系统中传播(数据派生出的对象也将会继承原有数据标签)，并使用这些标签来限制程序间的数据流向。机密性标签可以保护敏感数据不被非法或恶意用户的读取；而完整性标签可以保护重要信息或存储单元免受不可信或恶意用户的破坏。其广泛应用于与计算机系统安全相关的各个领域。

已有的工作在硬件层[2]、虚拟机层、高级语言层、低级语言层[3]、操作系统层[4]、网络层和数据层[5]实现了不同粒度的信息流控制，与本文密切相关的是虚拟机层实现和高级语言层实现，在虚拟机层，TaintDroid[6]在 Dalvik 虚拟机层实现了全系统动态污点跟踪，Trishul[7]基于 kaffe 虚拟机实现了 Java 动态污点跟踪，Laminar[8]实现了首个基于虚拟机和操作系统相结合的信息流跟踪控制系统，其他相关工作[9-13]在 JavaScript、Python、PHP 和 Java 上实现了信息流跟踪，但上述工作缺乏相关安全性分析与证明，且实现需要修改相应 Java 虚拟机运行环境，兼容性较差。Jif[14,15]基于编译器实现静态信息流分析与控制,并且实现了

**基金项目：**国家“863”计划资助项目（2015AA016006，2012AA012704）；国家重点研发计划项目（2016YFB0501900）

**作者简介：**吴泽智（1990-），男，湖南长沙人，博士研究生，主要研究方向为信息流控制、云计算安全（1141208772@qq.com）；陈性元（1963-），男，安徽无为人，教授，博导，博士，主要研究方向为网络与信息安全；杜学绘（1968-），女，河南新乡人，教授，博导，博士，主要研究方向为空间信息网络、云计算安全等；杨智（1975-），男，河南开封人，副教授，博士，主要研究方向为操作系统安全、云计算安全.

基于类型系统的无干扰安全性分析，但其实现需要修改编译器，且其大部分工作在静态完成，对动态信息流跟踪与控制支持不够。本文综合以上工作的优势，主要贡献如下：

a)提出了基于有限状态自动机的 Java 信息流分析方法，通过将整个程序变量污点取值空间抽象为状态空间，并 Java 字节码指令看做自动机状态转换动作，给出了自动机转换的信息流规则，为面向 Java 的信息流控制工作提供理论基础。

b)给出了基于高级输入和低级输出的无干扰安全的定义，证明了自动机状态转换的的无干扰安全性。

c)采用静态污点跟踪指令插入和动态污点跟踪与控制的方法实现了原型系统 IF-JVM，既不需要获得 Java 应用程序源码，也不需要修改 Java 编译器和解释器，更独立于客户操作系统。

## 1　自动机信息流分析

有限状态自动机(finite state automata)简称有限自动机，是具有离散输入输出系统的数学模型，广泛应用于程序语言的设计和实现中。同时，在安全模型形式化验证领域也有着重要的应用。它具有有限数量的状态，用此来记忆过去输入的有关信息，并根据当前的输入确定下一步的状态和行为。一个有限自动机可等价地看做一个系统行为(事件)驱动的状态转换图，系统的行为(事件)包括输入事件、输出事件和内部事件。因而信息流安全策略可以由系统状态转移规则来进行形式化描述。本文有限状态自动机具有两个功能，其一，用于记录系统所有实体及其安全标记的值和系统所有输入和输出动作的集合；其二，依据安全策略阻止信息流从高级实体向低级实体流动。本文自动机定义如下：

| | |
|---|---|
| $P$ | the name of a program |
| $v$ | the value of a varible |
| $s$ | the security context of a varible |
| $V$ | all variables in the program |
| $a$ | a set of actions |
| $\Omega$ | an execution or a sequence of actions |
| $\Pi$ | the programe counter of VM |
| $\Xi$ | the stack and heap of VM |
| $Q$ | the set of states of automata |
| $q_0$ | the start states of automata |
| $\chi$ | a finite set of input |
| $\kappa(\chi)$ | the secret inputs |
| $Ɗ(v)$ | the influenced varibles by the secret input |
| $\psi$ | a finite set of output |
| $\kappa(\psi)$ | the secret outputs |
| $T(v)$ | the security context of output varible |
| $\delta$ | a transition function |

$P$ 表示 Java 程序，程序由数据和指令组成。$v$ 表示变量的取值。变量的类型包括 boolean、byte、character、integer、short、long、float、double、object 和 arrays。$s$ 表示变量的安全标记值。为跟踪多源信息流，安全标记 $s$ 由不同种基本标记 $\wp$ 组成，安全标记构成偏序格( $\wp$，∧，⊥，T)，交汇运算∧取集合并集∪，对于任意安全标记 $x$ 和 $y$，有:（1）$x\wedge x=x$，满足等幂性;（2）$x\wedge y=y\wedge x$，满足可交换性;（3）$x\wedge(y\wedge z)=(x\wedge y)\wedge z$，满足可结合性.其顶元素是空集∅，表示为 T，对于 V 中的所有 $x$，有 T $\wedge x=x$;底元素是全集 U，为⊥，对于 V 中所有 $x$，有⊥$\wedge x=$⊥。$V(v,s)$表示该程序内所有的变量及其安全标记值。

$a$ 表示系统动作，可引起自动机状态发生转变。系统动作由不同类型的指令组成。如图 1 所示，本文将系统动作主要分为九类，分别是算术操作类( *math_op* )，初始化类( *allocate_op* )，返回操作类( *return_op* )，取数值类( *load_op* )，存数值类( *store_op* )，常数操作类( *const_op* )，跳转类( *jump_op* )，调用类( *invoke_op* )和栈操作类( *stack_op* )。对于算术操作类，其又包含 14 种子类型，以 add 指令 $a=a+b$ 为例，将程序中变量 $a$ 和 $b$ 数值 $v$ 进行加法运算并将结果保存于 $a$ 中，同时，将 $a$ 与 $b$ 的安全标记进行交汇运算∧，并更新 $a$ 的安全标记。对于初始化类，其中又包含 3 种子类型，以 newarray 为例，创建给定长度的数组同时创建相应安全标记。对于返回操作类，以 dreturn 为例，返回栈顶数值和相应的安全标记。对于存取操作类，在存取操作数同时存取其安全标记。对于常数操作类，规定常数安全标记取值为 0。对于后续操作类不再赘述。对于类似于无条件跳转 goto 和空值返回 return 等不改变自动机状态指令不列入系统动作。Ω 表示动作序列，由一系列顺序执行的动作组成。Π 表示程序计数器的安全标记值。Ξ 表示虚拟机堆栈存储的数据的安全标记值。由此可定义自动机状态及其状态转换。自动机状态定义为 $Q\,(V, sh, pc)$，自动机状态转换可描述为 $Q\,(V, sh, pc)\xrightarrow{a}Q'\,(V', sh', pc')$ 或者 $Q\,(V, sh, pc)\xrightarrow{\Omega}Q'\,(V', sh', pc')$。$q_0$ 表示自动机初始状态。

| | | |
|---|---|---|
| $a$ | ::= *math_op* | add \| mul \| div \| rem \| sub \| and \| or \| shl |
| | | \| shr \| ushr \| xor \| lcmp \| dcmpl \| dempg |
| | *allocate_op* | anewarray \| newarray \| multianewarray |
| | *return_op* | areturn \| dreturn \| ireturn \| freturn \| lreturn |
| | *load_op* | aaload \| baload \| caload \| daload \| iaload |
| | | \| faload \| laload \| saload \| dload \| fload \| iload |
| | | \| lload \| getfield \| getstatic |
| | *store_op* | aastore \| astore \| bastore \| castore \| dastore |
| | | \| iastore \| fastore \| lastore \| sastore \| dstore \| fstore |
| | | \| istore \| lstore \| putfield \| putstatic |
| | *const_op* | bipush \| sipush \| iconst \| lconst \| dconst \| fconst |
| | | \| lde \| ldew \| lde2_w \| aconst_null |
| | *jump_op* | if_acmpeq \| if_acmpne \| if_icmplt \| if_icmpge |
| | | if_icmple \| if_icmpeq \| if_icmpne \| ifeq \| ifne |
| | | ifgt \| ifge \| ifle \| iflt \| ifnennull \| ifnull |
| | *invoke_op* | invokespecial \| invokevirtual \| invokeinterface |
| | | \| invokestatic |
| | *stack_op* | swap \| pop \| pop2 |

图 1　系统动作类型

$\chi$ 表示程序输入集。程序输入包括用户键盘输入，读取文件，数据库或者网络接收数据等。$\kappa(\chi)$ 表示输入集中高级输入。例如用户输入的密码或程序读取的隐私文件等。$Ɗ(v)$ 表示在程序执行过程中受到高级输入影响的变量集合。$\psi$ 表示程序输出

集。程序输出包括屏幕显示(例如 printf()，写入文件，写入数据库或网络发送等。$\kappa(\psi)$ 表示输出集中高级输出，即输出内容中包含受高级输入影响的变量。$T(v)$ 表示输出变量的降密能力。例如，若变量 v 虽然受高级输入影响，但该高级输出为内部输出(如写入秘密文件)，为保证系统可用性与灵活性，该高级输出是安全的。$\delta$ 表示自动机状态转换函数，表示为 $Q\times\chi\cup\alpha\cup Q'\times\psi$。由此，自动机系统可完全描述为 $(Q, a, \chi, \psi, q_0)$。自动机状态转换规则如下：

$$
\begin{array}{ll}
(R-INIT) & q_0\ \Gamma:(v:\phi;\ s:\phi;\ \Pi:\phi;\ \Xi:\phi) \\
(R-INPUT-H) & Q\ \Gamma:(v:v_i;\ s:s_\phi;\ \Pi:D;\ \Xi:h)\xrightarrow{\kappa(\chi)}Q'\ \Gamma:(v:\kappa(\chi);\ s:(s_i\wedge s_{\kappa(\chi)});\ \Pi:D;\ \Xi:h) \\
(R-MATH) & Q\ \Gamma:(v:(v_i,v_j);\ s:(s_i,s_j);\ \Pi:D;\ \Xi:h)\xrightarrow{\vartheta_{math}}Q'\ \Gamma:(v:(v_i\vartheta v_j,v_j);\ s:(s_i\wedge s_j);\ \Pi:D;\ \Xi:h) \\
(R-ALLOCATE) & Q\ \Gamma:(v:\phi;\ s:\phi;\ \Pi:D;\ \Xi:h)\xrightarrow{allocate}Q'\ \Gamma:(v:v_\phi;\ s:s_\phi;\ \Pi:D;\ \Xi:h) \\
(R-RETURN) & Q\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:h)\xrightarrow{return}Q'\ \Gamma:(v:(v_i\wedge h);\ s:s_\phi;\ \Pi:D;\ \Xi:h) \\
(R-LOAD) & Q\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:h)\xrightarrow{load}Q'\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:(h\wedge s_i)) \\
(R-STORE) & Q\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:h)\xrightarrow{store}Q'\ \Gamma:(v:v_i;\ s:h;\ \Pi:D;\ \Xi:h) \\
(R-CONST) & Q\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:h)\xrightarrow{const}Q'\ \Gamma:(v:v_i;\ s:s_\phi;\ \Pi:D;\ \Xi:h) \\
(R-INVOKE) & Q\ \Gamma:(v:(v_i,s_\phi);\ s:(s_i,s_\phi);\ \Pi:D;\ \Xi:h)\xrightarrow{invoke}Q'\ \Gamma:(v:(v_i,v_i);\ s:(s_i,s_i);\ \Pi:D;\ \Xi:h) \\
(R-SWAP) & Q\ \Gamma:(v:(v_i,s_j);\ s:(s_i,s_j);\ \Pi:D;\ \Xi:h)\xrightarrow{invoke}Q'\ \Gamma:(v:(v_i,v_j);\ s:(s_i,s_j);\ \Pi:D;\ \Xi:h') \\
(R-JUMP) & Q\ \Gamma:(v:v_i;\ s:s_i;\ \Pi:D;\ \Xi:h)\xrightarrow{jump}Q'\ \Gamma:(v:v_i;\ s:h;\ \Pi:D;\ \Xi:h)\quad \text{if } D=\phi \\
(R-OUTPUT-L) & Q\ \Gamma:(v:v_\phi;\ s:s_\phi;\ \Pi:D;\ \Xi:h)\xrightarrow{\kappa(\psi)}Q'\ \Gamma:(v:\kappa(\psi);\ s:s_{\kappa(\psi)};\ \Pi:D;\ \Xi:h)\quad \text{if } s_{\kappa(\psi)}=\phi \\
(R-OUTPUT-H) & Q\ \Gamma:(v:v_\phi;\ s:s_\phi;\ \Pi:D;\ \Xi:h)\xrightarrow{\kappa(\psi)}Q'\ \Gamma:(v:\kappa(\psi);\ s:s_{\kappa(\psi)};\ \Pi:D;\ \Xi:h)\quad \text{if } s_{\kappa(\psi)}\le T(v)
\end{array}
$$

为便于理解，给出如表 1 所示程序代码及其自动机状态转换示意。对于第一条程序语句，应用 $R-ALLOCATE$ 和 $R-INPUT-H$ 规则，假设系统初始环境为空，$R-ALLOCATE$ 规则将变量 $a$ 及其安全标记添加到自动机状态中，$R-INPUT-H$ 规则根据输入 $\kappa(\chi)$ 设置变量 $a$ 的数值和安全标记值。同理，对于第二条程序语句执行相同的操作。对于第三条语句应用 $R-ALLOCATE$ 和 $R-CONST$ 规则，$R-ALLOCATE$ 规则将变量 $str$ 及其安全标记添加到自动机状态中，$R-CONST$ 规则设置常量的安全标记值为 $\phi$。对于第四条程序语句，应用 $R-INVOKE$ 规则，将实参数值和安全标记值传递给形式参数及其安全标记值。对于第八条程序语句，应用 $R-MATH$ 规则，将 $c$ 安全标记更新为 $c$ 与 $d$ 的安全标记值交汇值。对于第九条程序语句，应用 $R-RETURN$ 规则，将变量 $a$ 数值和安全标记设置为返回值和返回值的安全标记，并将该方法调用中使用的本地参数值及其安全标记从自动机状态中删除。对于第五条程序语句，应用 $R-OUTPUT-L$ 规则，变量 $str$ 其安全标记为空，因而该输出是安全的。对于第六条程序语句，应用 $R-JUMP$ 规则，变量 $b$ 其安全标记不为空，故该跳转是不安全的，自动机将拒绝该条程序语句和状态转换，第七条语句不能执行。

## 2　安全性证明

安全性证明采用无干扰分析方法，是一种抽象出安全本质的分析方法。直观可理解为：若受保护的数据与不受保护的数据之间不存在相互干扰，则认为该系统是安全的。在本文情形下，无干扰表现为程序高级输入不影响程序公开输出，即阻止了信息流从高级输入流向公开输出。无干扰安全可形式化定义为 $\forall(\kappa(\chi_1),\kappa(\chi_2))\quad \psi[\kappa(\chi_1)]\underline{\underline{EXE}}\psi[\kappa(\chi_2)]$。即对于任意不同类型的高级输入，系统公开输出是等价的。为证明系统满足无干扰安全性，首先给出安全标记传播的单调性定义及其证明。

安全标记传播的单调性定义为:①自动机状态一次转换由函数 $f$ 表示，对于安全标记格 $\wp$ 中的 $x$ 和 $y$，$f(x\wedge y)\le f(x)\wedge f(y)$。单调性也可等价定义为：②自动机状态一次转换由函数 $f$ 表示，对于安全标记格 $\wp$ 中的 $x$ 和 $y$，$x\le y$ 蕴含 $f(x)\le f(y)$。现证明两种定义是等价的：

先证明单调性②可推导出单调性①:由于 $x\wedge y$ 是 $x$ 和 $y$ 的最大下界,则 $x\wedge y\le x$ 且 $x\wedge y\le y$，由单调性②可知 $f(x\wedge y)\le f(x)$ 且 $f(x\wedge y)\le f(y)$，同时 $f(x\wedge y)$是 $f(x)$和 $f(y)$的最大下界，单调性①得证。然后，证明单调性①可推导出单调性②：假设 $x\le y$，由单调性①可知 $f(x\wedge y)\le f(x)\wedge f(y)$，根据定义有 $x\wedge y=x$。因此有 $f(x)\le f(x)\wedge f(y)$。因为 $f(x\wedge y)$是 $f(x)$和 $f(y)$的最大下界，得到 $f(x)\wedge f(y)\le f(y)$。从而 $f(x)\le f(x)\wedge f(y)\le f(y)$，即 $f(x)\le f(y)$，单调性②得证。

对于任意安全标记 $X$ 和安全标记 $Y$,$X$ 属于 $X\cup Y$。由此，可系统的无干扰安全性证明如下：

对于第一种情况：假设自动机初始状态 $q_0\ \Gamma:(v:\phi;\ s:\phi;\ \Pi:\phi;\ \Xi:\phi)$，一个状态转换序列 $\Omega$ 可分解为 n 个动作 $a_1, a_2, \dots, a_n$。$a_n$ 为公开输出动作，由规则 $R-OUTPUT-L$ 可知该公开输出动作 $\kappa(\psi)$ 为空。$a_1$ 为某高级输入动作 $\kappa(\chi_1)$，执行后自动机状态转换为 $Q\ \Gamma:(v:v_{\kappa(\chi_1)};\ s:s_{\kappa(\chi_1)};\ \Pi:D;\ \Xi:h)$。$a_2$，…，$a_{n-1}$ 为任意输入输出或者其他动作。假设系统高级输入动作影响了系统公开输出动作，则有，执行动作 $a_n$ 后，自动机状态由 $Q\ \Gamma:(v:v_{a_{n-1}};\ s:s_{a_{n-1}};\ \Pi:D;\ \Xi:h)$ 转换为 $Q\ \Gamma:(v:v_{a_n};\ s:s_{a_n};\ \Pi:D;\ \Xi:h)$。由单调性可知，有 $s_{a_{n-1}}\le s_{a_n}$，由于公开输出动作 $\kappa(\psi)$ 为空，即 $s_{a_n}$ 为空，故 $s_{a_{n-1}}$ 也为空。执行动作 $a_{n-1}$ 后，自动机状态由 $Q\ \Gamma:(v:v_{a_{n-2}};\ s:s_{a_{n-2}};\ \Pi:D;\ \Xi:h)$ 转换

为 $Q\,\Gamma:(v:v_{a_{n-1}};\ s:s_{a_{n-1}};\ \Pi:D;\ \Xi:h)$。由单调性可知，有 $s_{a_{n-2}}\le s_{a_{n-1}}$，由于 $s_{a_{n-1}}$ 为空，故 $s_{a_{n-2}}$ 也为空。依次类推，可知 $s_{a_1}$ 为空，即 $s_{\kappa(\chi_1)}$ 为空，而显然 $s_{\kappa(\chi_1)}$ 不为空。与前提矛盾，故系统高级输入动作不影响系统公开输出动作。即对于任意不同类型的高级输入，系统公开输出的安全标记皆为空。

上述无干扰性只考虑了系统公开输出(一般保证了系统公开输出无干扰安全，则系统是安全的)，若该输出为系统内部输出，系统仍满足特定条件下的无干扰安全性：$\forall(s_{\kappa(\chi_1)}\le T(v),s_{\kappa(\chi_2)}\le T(v))\quad \psi[\kappa(\chi_1)]\underline{\underline{\mathrm{EXE}}}\psi[\kappa(\chi_2)]$。即任意高级输入与降密能力满足偏序关系，系统内部输出是等价的。证明如下：

假设自动机初始状态 $q_0\ \Gamma:(v:\phi;\ s:\phi;\ \Pi:\phi;\ \Xi:\phi)$，一个状态转换序列 $\Omega$ 可分解为 $n$ 个动作 $a_1$，$a_2$，…，$a_n$。$a_n$ 为内部输出动作，由规则 $R-OUTPUT-H$ 可知该内部输出动作 $s_{\kappa(\psi)}\le T(v)$。$a_1$ 为某高级输入动作 $\kappa(\chi)$，执行后自动机状态转换为 $Q\,\Gamma:(v:v_{\kappa(\chi)};\ s:s_{\kappa(\chi)};\ \Pi:D;\ \Xi:h)$。$a_2$，…，$a_{n-1}$ 为任意输入输出或者其他动作。执行动作 $a_n$ 后，自动机状态由 $Q\,\Gamma:(v:v_{a_{n-1}};\ s:s_{a_{n-1}};\ \Pi:D;\ \Xi:h)$ 转换为 $Q\,\Gamma:(v:v_{a_n};\ s:s_{a_n};\ \Pi:D;\ \Xi:h)$。由单调性可知，有 $s_{a_{n-1}}\le s_{a_n}$，由于内部输出动作 $s_{\kappa(\psi)}\le T(v)$，即 $s_{a_n}\le T(v)$，故 $s_{a_{n-1}}\le T(v)$。执行动作 $a_{n-1}$ 后，自动机状态由 $Q\,\Gamma:(v:v_{a_{n-2}};\ s:s_{a_{n-2}};\ \Pi:D;\ \Xi:h)$ 转换为 $Q\,\Gamma:(v:v_{a_{n-1}};\ s:s_{a_{n-1}};\ \Pi:D;\ \Xi:h)$。由单调性可知，有 $s_{a_{n-2}}\le s_{a_{n-1}}$，由 $s_{a_{n-1}}\le T(v)$ 可知 $s_{a_{n-2}}\le T(v)$。依次类推，可知 $s_{a_{i-1}}\le s_{a_i}$，$s_{a_{i-1}}\le T(v)$。即有 $s_{a_1}\le T(v)$，$s_{\kappa(\chi)}\le T(v)$。假设系统满足偏序关系的高级输入动作 $\kappa(\chi_1)$ 与 $\kappa(\chi_2)$ 影响了系统内部输出动作，则必有，$(s_{\kappa(\chi_1)}\le T(v))\wedge(s_{\kappa(\chi_2)}\not\subset T(v))$ 或者 $(s_{\kappa(\chi_1)}\not\subset T(v))\wedge(s_{\kappa(\chi_2)}\le T(v))$。而与已知条件 $(s_{\kappa(\chi_1)}\le T(v))\wedge(s_{\kappa(\chi_2)}\le T(v))$ 矛盾。故满足偏序关系的高级输入动作 $\kappa(\chi_1)$ 与 $\kappa(\chi_2)$ 不影响系统内部输出。

表 1　自动机状态变迁示例

| 程序代码 | 自动机输入 | 自动机判断 | 自动机状态 |
|---|---|---|---|
| public class example{ | allocate(*a*) | OK | {*a*}:{ $\phi$ } |
| 1.　int *a*=system.in(); | input(*a*) | OK | {*a*}:{ $s_1$ } |
| | allocate(*b*) | OK | {*a, b*}:{ $s_1$ , $\phi$ } |
| 2.　int *b*=system.in(); | input(*b*) | OK | {*a, b*}:{ $s_1$ ,$s_2$ } |
| | allocate(*str*) | OK | {*a, b, str*}:{ $s_1$ ,$s_2$ , $\phi$ } |
| 3．String *str*='hello'; | const(*str*) | OK | {*a, b, str*}:{ $s_1$ ,$s_2$ , $\phi$ } |
| 4．*a*=plus(*a, b*); | invoke(plus) | OK | {*a, b, str, c, d*}:{ $s_1$ ,$s_2$ , $\phi$ , $s_1$ ,$s_2$} |
| 5．system.out(*str*); | output(*str*) | OK | {*a, b, str*}:{ $s_1\wedge s_2$, $s_2$ , $\phi$ } |
| 6．if(*b*) | jump(*b*) | NO | - |
| 7.　system.out(*str*); | - | - | - |
| plus(int *c*, int *d*) { | - | - | - |
| 8．*c*=*c*+*d*; | math(add) | OK | {*a, b, str, c, d*}:{ $s_1$ ,$s_2$ , $\phi$ , $s_1\wedge s_2$, $s_2$} |
| 9．return *c*;} | return(*c*) | | {*a, b, str*}:{ $s_1\wedge s_2$, $s_2$ , $\phi$ } |
| } | | | |

## 3　系统实现与测试

### 3.1　系统实现

系统实现采用动态污点跟踪技术，结合静态污点跟踪指令插入和动态污点跟踪与控制的方法，实现了对 Java 虚拟机细粒度地信息流跟踪与控制。既不需要获得 Java 应用程序源码，也不需要修改 Java 解释器，更独立于客户操作系统。

为实现细粒度数据标记，需要为每个程序变量设置相应污点影子变量。为便于理解，给出源代码层次污点跟踪代码插入示例，如表 2 所示。污点变量类型为 taint，可针对不同跟踪粒度和跟踪需求进行相应设置，本文定义 taint 为 32 位无符号整数。程序变量包括基本变量类型和引用类型，基本类型如 int *a*，double *b* 为其设置污点 *a_tag* 与 *b_tag*。引用类型包括对象和数组，对象包括实例化的类，为其设置污点 class_tag；数组又包括基本类型数组和对象类型数组，基本类型数组 byte[] *array* 为设置污点 taint[] *arrar_tag*，每个数组值对应相应污点值，对象类型数组污点值保存于相应对象中。Java 虚拟机栈包括两部分：操作栈和局部变量区，该两部分可看做一个方法帧，对于方法调用 plus(*c*, *d*)，需要创建一个新的方法帧，并传递参数及其污点 plus(*c*, *c_tag*, *d*, *d_tag*)。当调用结束时，需要返回相应的数值及其污点标记。因而，在非空返回时，需要将数值及其污点存入容器内并返回 return taintint.valueof(*c*, *c_tag*)，返回后销毁该帧。

表 2　污点跟踪代码插入示例

| 原始程序代码 | 污点跟踪程序代码 |
|---|---|
| public class example{ | public class example{ |
| | taint *class_tag*; |
| int *a*; | int *a*; |
| | taint *a_tag*; |
| double *b*; | double *b*; |
| | taint *b_tag*; |
| byte[] *array*; | byte[] *array*; |
| | taint[] *arrar_tag*; |
| *array*= new byte[5]; | *array*= new byte[5]; |
| | *array*= new taint[5]; |
| plus(int *c*, int *d*) { | plus(int *c*, taint *c_tag*, int *d*, taint *d_tag*) { |
| *c*=*c*+*d*; | *c*=*c*+*d*; |
| | *c_tag*=*c_tag* \| *d_tag*; |
| return *c*;} | return taintint.valueof(*c*, *c_tag*)} |
| } | } |

为实现细粒度信息流跟踪，需要为每种可发生污点传播的指令进行污点传播。其具有代表性的指令如表 3 所示。

为实现细粒度信息流控制，需要在程序关键调用点进行数据污点标记和数据污点检测。用户可使用配置文件预定义某类方法返回值污点类型(如用户键盘输入)及方法参数的污点检测

(如执行 SQL 语句)。同时，用户可使用 SetTaint(*val*,*taint*)和 GetTaint(*val*)来设置和获取任意变量的污点标记。

表 3　指令级污点传播分析

| 序号 | 指令类型 | 指令含义 | 污点传播操作 |
|---|---|---|---|
| 1 | 算术(add) | 将栈顶两数相加 | 将栈顶两数污点标记交汇运算后更新栈顶数污点标记 |
| 2 | 分配(newarray) | 创建指定长度的数组 | 将数组长度变量污点设置为空，并创建相应长度的污点影子数组 |
| 3 | 取数 1(iload) | 从本地变量取数入栈 | 将相应本地变量的污点影子变量取到栈顶下 |
| 4 | 取数 2(iaload) | 从数组中取数入栈 | 将该数索引值污点设置为空，并将其污点影子数组相应索引位置取入栈顶下 |
| 5 | 存数 1(istore) | 将数据存入本地变量 | 将相应本地变量的污点影子变量存入 |
| 6 | 存数 2(iaload) | 将数据存入数组 | 将该数索引值污点设置为空，并将其污点存入相应索引位置污点影子数组 |
| 7 | 返回 1(ireturn) | 将方法的返回值于栈顶返回 | 将返回值及其污点值存入返回容器，执行 areturn |
| 8 | 返回 2(areturn) | 将方法的引用对象于栈顶返回 | 若该栈顶对象是基本类型数组，存入其数值与污点 |
| 9 | 常数(iconst) | 取常数入栈 | 将相应污点影子变量设置为空 |
| 10 | 跳转 1(ifge) | 比较栈顶整数并跳转 | 栈顶弹出相应的污点影子变量 |
| 11 | 跳转 2(if_icmpge) | 比较栈顶两整数并跳转 | 栈顶弹出相应的两污点影子变量 |
| 12 | 交换(swap) | 交换栈顶两数 | 交换两数相应的污点数值 |
| 13 | 弹出(pop) | 栈顶弹出 | 若该数是基本类型或基本类型数组，弹出其污点影子变量 |
| 14 | 调用(invoke) | 调用一个方法，弹出调用参数并压入返回值 | 将该方法转换为含污点参数的调用方法，若参数对象通过基本类型数组传递，将其存入容器 |
| 15 | 长度(arraylength) | 计算数组长度 | 在栈顶返回值下存入数值‘0’ |

### 3.2　系统测试

系统测试环境为：os/Ubuntu-12.04, CPU/2.27 GHz, RAM/4 GB,使用虚拟机版本为 OpenJDK “IcedTea” JVM 1.8.0。选择测试工具 Scalabench。

第一，测试两系统的运行时间，每个子测试项目运行 10 次并记录平均值(子测试项目详细信息可参见：http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite/)，其结果如图 2 所示。IF-JVM 较 JVM 运行效率平均延迟 53.1%，这是由于除了运行程序自身指令之外，还需要运行静态插入的污点传播指令。其中子测试项目 scaladoc 性能延迟最大，这是由于该测试程序中对于数组及字符串等对象处理指令过多，导致污点传播指令增多。

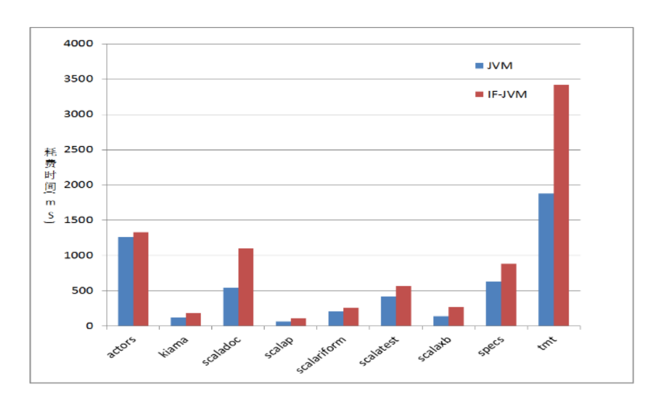



图 2　两系统运行时间对比

第二，测试两系统在运行时最大堆的内存占用，其结果如图 3 所示。IF-JVM 较 JVM 平均多占用堆内存 323.1%，这是由于除了程序自身数据结构需要占用堆内存之外，静态插入的用于保存相应数据的污点数据结构也需要占用堆内存。

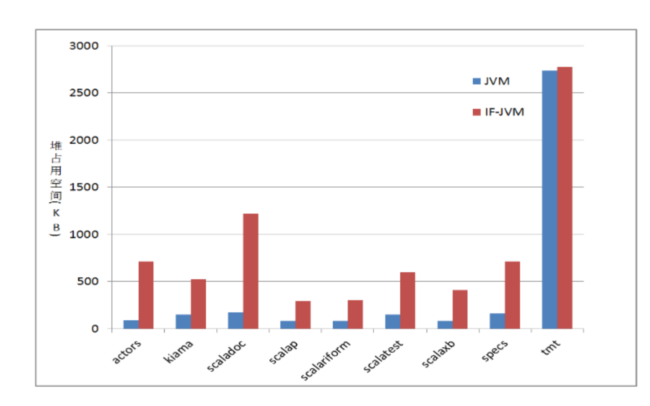



图 3　两系统堆空间占用对比

## 4　结束语

针对当前面向 Java 的信息流分析工作需要修改编译器或实时执行环境，对已有系统兼容性差且缺乏形式化分析与安全性证明等不足，提出了基于有限状态自动机的 Java 信息流分析方法，并证明了程序执行的无干扰安全性。采用静态污点跟踪指令插入和动态污点跟踪与控制的方法实现了原型系统 IF-JVM，实验结果表明原型系统能正确实现了对 Java 虚拟机细粒度地信息流跟踪与控制。后续可研究插桩系统性能优化，使得插入的污点指令更加精简高效。

## 参考文献：

[1] 吴泽智, 陈性元, 杨智, 等. 信息流控制研究进展 [J]. 软件学报, 2017, 28 (1): 135-159.

[2] Crandall J R, Chong F T. Minos: Control data attack prevention orthogonal to memory model [C]// Proc of the 37th Intetnational Symp. on Microarchitecture. LA: IEEE Press, 2004: 221-232.

[3] Kemerlis V P, Portokalidis G, Jee K, Keromytis A D. Libdft: Practical dynamic data flow tracking for commodity systems [J]. ACM SIGPLAN Notices, 2012, 47 (7): 121-132.

[4] Krohn M, Yip A, Brodsky M, et al. Information flow control for standard OS abstractions [C]// Proc of ACM SIGOPS Operating Systems Review. New York: ACM Press, 2007: 321-334.

[5] Schultz D, Liskov B. IFDB: decentralized information flow control for databases [C]// Proc of the 8th ACM European Conference on Computer Systems. New York: ACM Press, 2013: 43-56.

[6] Enck W, Gilbert P, Chun B G, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones [C]// Proc of OSDI. Berkeley: USENIX Association, 2010: 255-270.

[7] Nair S K, Simpson P N D, Crispo B, et al. A virtual machine based information flow control system for policy enforcement [J]. Electronic Notes in Theoretical Computer Science, 2008, 197 (1): 3-16.

[8] Roy I, Porter D E, Bond M D, et al. Laminar: practical fine-grained decentralized information flow control [J]. ACM SIGPLAN Notices—PLDI, 2009, 44 (6): 63-74.

[9] Haldar V, Chandra D, Franz M. Dynamic taint propagation for Java [C]// Proc of Computer Security Applications Conference. 2006: 301-311.

[10] Matej V, Binder W, Hauswirth M. ShadowData: shadowing heap objects in Java [C]// Proc of ACM Sigplan-Sigsoft Workshop on Program Analysis for Software TOOLS and Engineering. New York: ACM Press, 2013: 17-24.

[11] Chandra D, Franz M. Fine-Grained information flow analysis and enforcement in a Java virtual machine [C]// Proc of the 23rd Annual Computer Security Applications Conference. New York: IEEE Press, 2007: 463-475.

[12] Manivannan K, Wimmer C, Franz M. Decentralized information flow control on a bare-metal JVM [C]// Proc. of the 6th Annual Workshop on Cyber Security and Information Intelligence Research. New York: ACM Press, 2010: 64-74.

[13] Myers A C, Liskov B. Protecting privacy using the decentralized label model [J]. ACM Trans on Software Engineering and Methodology, 2000, 9 (4): 410-442.

[14] Myers A C. JFlow: practical mostly-static information flow control [C]// Proc of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1999: 228-241.

[15] Blackburn S M, Garner R, Hoffmann C, et al. The dacapo benchmarks: Java benchmarking development and analysis [C]// Proc of OOPSLA. New York: ACM Press, 2006: 169-190.